Theoretical Computer Science 72 (1990) 225-249
North-Holland

IMPLEMENTING TERM REWRITE LANGUAGES IN DACTL

Richard KENNAWAY

School of Information Systems, University of East Anglia, Norwich, UK NR4 7TJ

Abstract. Dactl is a low-level language of graph rewriting, intended for programming highly parallel machines. The language includes, but is not restricted to, the limited form of graph rewriting which is commonly used to implement functional language such as Miranda, ML, Hope and Clean. In contrast to these functional languages, where the order in which subterms are evaluated (the *evaluation strategy*) is fixed for all programs, in Dactl the evaluation strategy is programmed explicitly. We define a translation of a functional language into Dactl, describe the problems encountered and their solution, and prove that the translation is correct.

1. Term rewrite languages for functional programming

There are several programming languages in which programs are term rewrite systems. Miranda[1] [20], Hope [8], FP [1], ML [17] and Clean [6] are examples. In these languages, a program consists of a set of rewrite rules defining constants and functions, and an expression, also called a term, to be evaluated according to those rules. In this paper, we assume some familiarity with the concepts of term rewrite systems, such as redex, reduction relation, normal form, regular rewrite system, etc. A good reference is [18].

Here is a simple example of a term rewrite program, written in Miranda. It defines the well-known factorial function.

$$\text{fac } 0 = 1$$

$$\text{fac } n = n * (\text{fac}(n-1))$$

$$\text{fac } 10.$$

The term (fac 10) is evaluated by applying the two rules which define the function fac, and the following computation ensues:

$$\begin{aligned}\text{fac } 10 &= 10 * (\text{fac}(10-1)) = 10 * (\text{fac } 9) = 10 * (9 * (\text{fac}(9-1))) = \cdots \\ &= 10 * (9 * (8 * (7 * (6 * (5 * (4 * (3 * (2 * (1 * 1)))))))))) = \cdots = 3628800.\end{aligned}$$

There are some subtleties that may not be apparent at first sight.

(1) There are built-in arithmetic operators, notionally defined by an infinite set of rules such as $1+1=2$, $5-1=4$, $6*9=54$, etc.

[1] Miranda is a trademark of Research Software Ltd.

(2) The term (fac 0) is considered to match only the first rule, not the second. In Miranda, if several rules match a term, only the first of the matching rules may be executed. This reliance on textual order is convenient for the programmer, though it causes certain problems for a formal definition of the semantics.

(3) The term (fac(10 − 1)) is not considered to match the second fac rule, because until the subterm (10 − 1) is evaluated, it is not known whether the term might in future match the first tule.

(4) Miranda is *lazy*—programs may operate on infinite data structures, without causing non-termination. This is a point on which term rewrite languages differ. Hope, ML, and FP are strict, while Clean is lazy.

2. The Dactl language

Dactl [12, 13] is a language of graph rewriting, designed to be a low-level language for highly parallel machines. Here is the factorial example written in Dactl:

```
Fac[0]⇒*1|
Fac[n:(INT−0)]⇒#IMul[n^#Fac[^*ISub[n 1]]]|
Initial⇒*Fac[10].
```

Apart from various trivialities of concrete syntax (argument-lists enclosed in "[ ]", "⇒" instead of "=", separation of rules by "|", no special syntax for arithmetic expressions, etc.) there are several more fundamental differences between Dactl and the languages discussed above.

2.1. Graphs, not trees

While graph rewriting is a common implementation technique for functional languages, it is possible to implement these languages by string reduction. Dactl, in contrast, explicitly talks about graphs. Graph reduction is an essential part of the semantics of Dactl, not an optimisation.

The right-hand side of a Dactl rule denotes a graph. When there are no repeated variables, this graph is essentially the syntax tree. Repeated variables imply identity of nodes, yielding a graph rather than a tree. The right-hand side of the second Fac rule denotes the graph of Fig. 1. The meanings of the "*", "#" and "^" marks are explained below. The graph to be evaluated by the program is specified by means of the reserved function symbol Initial. The initial state of the graph is always

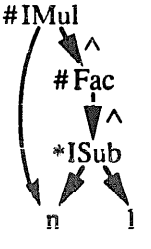


Fig. 1.

*Initial, and the graph which the programmer wishes to evaluate is specified as the right-hand side of a rule for Initial.

2.2. The evaluation strategy

Each of the functional languages we mentioned in Section 1 has its own *evaluation strategy*, which determines the order in which subterms may be reduced. That strategy varies from one language to another, and is reponsible for the fact that some languages are lazy and some are strict. But for each language, the strategy is fixed.

Such a strategy is usually driven by pattern-matching. We saw in Section 1 that when evaluating the term fac(10−1), the attempt to match this against the first rule for fac tries to match the subterm (10−1) against 0. As (10−1) has an operator as its principal function symbol, this is taken as indicating that this subterm must be evaluated to "constructor" form before continuing with the attempt to match.

Dactl takes a completely different approach. The pattern-matching of Dactl rules is a purely passive process: it *never* invokes evaluation. Instead, the evaluation strategy is programmed explicitly, by attaching control marks to the nodes and edges of the right-hand sides of the rules.

The basic idea is that at each moment, only certain nodes in the graph are candidates for evaluation. These nodes are called *active*. Some nodes may be *blocked*, waiting for some of their argument nodes to be evaluated before they can become active. The remaining nodes are *idle*; they are not candidates for evaluation, nor are they waiting for other nodes to be evaluated. These states are indicated by the markings "*" (active), "#...#" (blocked, waiting for as many notifications as there are #s), and no mark (idle). The out-arcs from a node to those of its arguments on which it is waiting bear the *notify* mark "^"; other arcs bear no mark. For the purpose of discussing the language, the idle and non-notify marks may be denoted by □ and △, respectively.

Rule-matching and -execution may be attempted at any active node. If there are many active nodes, it may be attempted at any set of them simultaneously. This is the source of parallelism in Dactl. Control markings play no other part in rule-matching.

If rule-matching is attempted at some active node, but it is found that no rule of the system matches, then the node is made inactive, and each of its parents to which it is joined by a notify arc is notified. The notify marks are removed, and the blocked counts of those parents are reduced by one. Wherever this brings the count to zero, that parent is made active.

If rule-matching succeeds, then the rule which matched (or one of them, if there is more than one) is executed.

(i) The root of the match is made idle.

(ii) A new copy of the right-hand side of the rule is added to the graph, in which references to free variables of the left-hand side are replaced by references to the nodes to which they were matched. The right-hand side of the rule specifies not only the function symbols on the nodes, but also the control markings.

(iii) Where the right-hand side specifies a mark on a left-hand side variable, that mark is attached to the node the variable was bound to, if that node is currently idle. (If the node is active or blocked, its mark is not changed.)

(iv) All references in the graph to the root of the matched subgraph are replaced by references to the root of the copy of the right-hand side.

In practice, the potentially impractical redirection of all references to the redex-root may usually be implemented by overwriting the redex-root with the contents of the root of the copy of the right-hand side. However, this is only a useful implementation technique; redirection is what the formal semantics stipulates. Computation terminates when there are no active nodes in the graph.

We may say that Dactl has, not an evaluation strategy, but a meta-strategy: every active node is a candidate for rule-matching and -execution. Within that meta-strategy, the user programs particular strategies by his assignment of control markings to the right-hand sides of his rules. The "user" will normally be a compiler, not a person.

Dactl rules are in fact more general than our description here; for example, a rule can perform more than one redirection; and the number of # marks on a node need not be the same as the number of out-arcs bearing the notify mark. These more general forms allow the translation of logic languages and imperative languages, as well as functional languages. We have only described the subset which we need for the purposes of this paper.

Readers acquainted with [11] should note that paper deals with Dactl0, a predecessor of Dactl.

2.3. Example of a computation

Figure 2 illustrates some stages in the factorial computation. We omit nodes which are not accessible from the root of the graph; in practice, they will be garbage collected only when memory is exhausted.

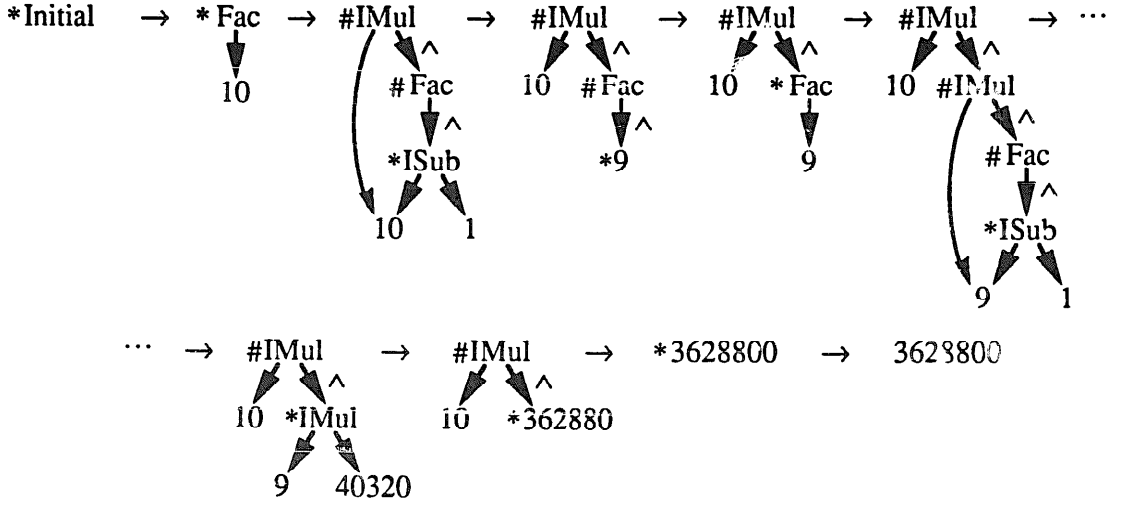


Fig. 2.

2.4. Unordered rules

We saw that Miranda depends on textual ordering to disambiguate the two rules for fac. Dactl rule-systems are unordered. If the left-hand side of the second rule were replaced by Fac[n:INT], then the graph *Fac[0] would match both the first and second rules, and either could be chosen. In the pattern Fac[n:(INT−0)], the "−" is a *pattern operator*: $P-Q$ by definition matches any graph which matches P and not Q. This makes rule-ordering unnecessary.

In fact, Dactl does include a syntax for expressing rule-ordering, but only as syntactic sugar; its semantics is defined by translation to unordered rules and pattern operators.

3. Terminology for rewrite systems

3.1. Term rewrite systems

We assume familiarity with the concepts of term rewriting, such as redex, normal form, regular TRS, etc., and will only state our notations. Variables and function symbols are alphanumeric strings, variables beginning with a lower case letter and function symbols beginning with an upper-case letter or a digit. A term t is either a variable x, or has the form $F[t_1 \ldots t_n]$ $(n \geq 0)$; when $n=0$ we omit the brackets. When we do not need to indicate n, we may also write this as $F[\underline{t}]$. F is the *principal function symbol* of this term. A term is *closed* if it contains no variables.

The ith argument of a term t is denoted by $t{\cdot}i$. For example, Cons[x Cons[y Cons[z Nil]]]·2·2·1 is the term z. The string 2·2·1 is an *address* of the term. If every address of t is an address of t', and every address of a function symbol of t is an address of a function symbol of t', then t is a *prefix* of t'.

A *rewrite rule* R has the form $t_1 \Rightarrow t_2$, where every variable appearing in t_2 also appears in t_1. t_1 may be denoted by *left*(R), and t_2 by *right*(R). It is a *rule for* the principal function symbol of t_1. A *rule-system* is a set of rules. We shall exclude for the moment notations like Fac[(INT−0)], regarding them as merely a convenient shorthand for expressing a large set (in this case, an infinite set) of rules.

A *substitution* is a function from some set of variables to terms. The result $\sigma(t)$ of applying a substitution σ to a term t is obtained by replacing every occurrence in t of every variable x in the domain of σ by $\sigma(x)$. $\mathrm{Red}_T(t, t')$ means that there is a rule $t_1 \Rightarrow t_2$ of T and a substitution σ such that $t=\sigma(t_1)$ and $t'=\sigma(t_2)$. t is a *redex* of the rule and t' its *reduct*. $t \rightarrow_T t'$ means that t may be transformed to t' by replacing some redex contained in t by its reduct. $\rightarrow_T^*$ is the reflexive transitive closure of $\rightarrow_T$. A term containing no redexes is in *normal form*.

A *term rewrite system* T is a set of symbols Funct(T), a set of rules Rules(T) using only those symbols, and a set of terms Terms(T) over those symbols which is closed under reduction by the rules and by the subterm relation. NF(T) denotes the set of normal forms of T. We do not assume that Terms (T) must contain all the terms which can be formed from Funct(T). This allows us to uniformly handle

systems with or without some type structure. From our point of view, a type system merely limits the membership of Terms(T).

A function symbol F of a rewrite system is an *operator* if there is a rule for F. Otherwise it is a *constructor*. A term is in *constructor form* if its principal function symbol is a constructor.

If $F[t_1 \ldots t_n]$ is the left-hand side of a term rule R and $1 \leqslant i \leqslant n$, we say that R *pattern-matches at the ith place* if t_i is not a variable. If every rule for F pattern-matches at the ith place, then i is an *always-matched* place of F. If no rule for F does so, then i is a *never-matched* place of F. Otherwise, i is a *sometimes-matched* place of F.

3.2. Graph rewrite systems

Dactl is a language of graph rewriting, and graph rewriting is the usual technique for implementing term rewriting. Some properties of graph rewriting are simpler than the corresponding features of term rewriting. For these reasons, we shall primarily consider not term rewriting as such, but its graphical implementation, *term graph rewriting*. We shall now describe the differences in concepts and notations.

A term graph t is a rooted directed graph in which each node is labelled with either a variable or a function symbol. It is *closed* if it contains no variables. A node labelled with a variable has no out-arcs, and no variable may occur in the graph more than once. The out-arcs of a node are ordered. Every node must be accessible from the root. The *descendants* of a node are the nodes at the other ends of its out-arcs. The *arguments* of a term graph are the term subgraphs rooted at the descendants of its root. A term may be considered as a graph, its syntax tree, modified by coalescing variable nodes bearing the same variable. To write more complicated graphs in a textual form, we can use Dactl syntax. Thus Plus[x:Times[2 3]x] denotes the graph of Fig. 3. (Note that here x does not denote a variable node, but simply expresses the graph structure, and is not present in the actual graph.) There is a natural correspondence between the nodes of a term graph and the sub-term graphs rooted at those nodes, and we may use the two interchangeably.

When we describe a graph as of the form $F[t_1 \ldots t_n]$, the subgraphs $t_1 \ldots t_n$ may share nodes. Thus when we write the graph as $F[\underline{t}]$, $\underline{t}$ denotes not a list of graphs, but a single graph with a list of n roots.

The concepts of address and prefix carry over unchanged. For a rewrite rule, we stipulate that the left- and right-hand sides of the rule are disjoint term graphs. A

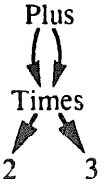


Fig. 3.

term graph is *linear* if no arc points to its root, and no more than one arc points to any other node. (That is, the graph is a tree.)

A *substitution* is now a function which maps a set of variables $\underline{x}$ to the roots of a multi-rooted graph $\underline{t}$. The result $\sigma(t)$ of applying a substitution σ to a term graph t is obtained by making a copy of $\underline{t}$, replacing every arc in t to a variable x in the domain of σ by an arc to the copy of $\sigma(x)$, and removing all nodes in the new graph inaccessible from the root. If the root of t is itself a variable x in the domain of σ, then the root of the new graph is the copy of $\sigma(x)$. This gives definitions of $\mathrm{Red}_T(t, t')$ and $t \to_T t'$ for term graphs analogous to those for terms.

Real implementations of term graph rewriting do not perform the redirection we have just described, but instead overwrite the root of the redex with the function symbol and descendant-pointers of the root of the reduct. Where there is no such root (if the right-hand side is just a variable) the technique of "indirection" nodes is used. We consider these to be implementation techniques below the level at which we are working. From our point of view, rewriting is always performed by redirection, and indirection nodes never appear.

A *term graph rewrite system* (or TGRS) T is a set of symbols $\mathrm{Funct}(T)$, a set of rules $\mathrm{Rules}(T)$ using only those symbols, and a set of term graphs $\mathrm{TG}(T)$ over those symbols, closed under reduction by the rules and by the subgraph relation.

The remaining definitions of Section 3.1 carry over unchanged. Every regular TRS T can be read as a TGRS which we denote by T^{G}. The graphs constituting the rules of T^{G} are the syntax trees of the terms in the rules of T, except that where the right-hand side of a rule of T has repeated variables, the right-hand side of the corresponding rule of T^{G} has multiple arcs pointing to a single node bearing that variable. $\mathrm{TG}(T^{\mathrm{G}})$ is the set of all graphs which "unravel" in the obvious sense to terms in $\mathrm{Terms}(T)$.

Barendregt et al. [3] discuss in detail the relation between TRSs and their corresponding TGRSs. A term in a regular TRS has a normal form iff every graph in the corresponding TGRS which unravels to that term has a normal form. There is an obvious correspondence between the reduction sequences, each graph redex corresponding to a set of isomorphic term redexes.

4. Simple term graph rewrite systems

We shall define a translation from a class of TRSs into Dactl. The class is defined by the following conditions.

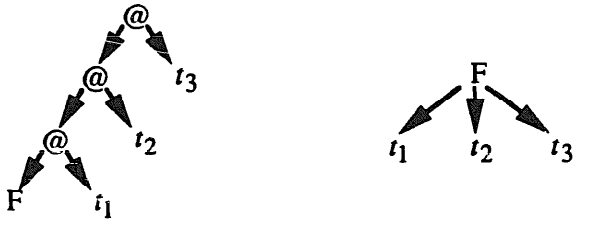
4.1. The rules must be unordered

We have seen that textual ordering is used by Miranda for disambiguating overlapping rules. Having priorities between rules poses certain deep problems in defining the precise semantics of the rewrite system (explored e.g. in [4]). However, Lavilie [16] has shown how the use of textual ordering in ML may be eliminated

by a transformation into an equivalent unordered rewrite system. His methods apply to other rewrite-based functional languages as well. We therefore do not lose generality in only considering unordered systems.

4.2. The rewrite system must be functional, not applicative

The distinction we intend by these words is this. In some term rewrite languages, all functions are "curried"; a term written as $(F\,t_1\,t_2 \ldots t_n)$ has two immediate subsystems: $(F\,t_1\,t_2 \ldots t_{n-1})$ and t_n. In effect, F is a nullary function symbol, attached to its arguments by an invisible, left-associative, binary "apply" operator. Looked at in this way, "apply" is both an operator and a constructor, which we will disallow below. Instead, we apply F to the whole tuple of arguments at once, writing $F[t_1\,t_2 \ldots t_n]$, where the immediate subterms are each of $t_1, t_2, \ldots, t_n$. The distinction is made clear by the syntax trees of Fig. 4 (which is what we consider these textual representations to denote).



Applicative Functional

Fig. 4.

At first sight, this appears to exclude programming with higher-order functions, such as this Miranda example. (Miranda is applicative.)

$$\text{map}\,f\ \text{Nil} = \text{Nil}$$

$$\text{map}\,f(\text{Cons}\,x\,y) = \text{Cons}(f\,x)(\text{map}\,f\,y).$$

However, it is possible to transform applicative systems into operator-constructor form by replacing each operator F of applicative arity n by a set of operators $F_0 = F, \ldots, F_{n-1}, F_n$, of arities $0 \ldots n$, giving the application operator Ap all rules of the form

$$\text{Ap}[F_i[x_1 \ldots x_i]\,x] \Rightarrow F_{i+1}[x_1 \ldots x_i\,x] \quad (i{:}0 \ldots n-1)$$

and giving F_n the original rules for F, with the left-hand side written in functional form. Thus the applicative notation can be seen as syntactic sugar applied to an underlying functional system.

4.3. Each function symbol must have a fixed arity

Each use of any given function symbol must apply that symbol to the same number of arguments. This is only a convenience, as uses of the same function symbol with

different arities can be considered as uses of different symbols, one different symbol for each different arity.

4.4. The rewrite system must be left-linear

That is, the left-hand side of every rule must contain no repeated variables. (For term graph rewrite systems, the corresponding notion is that the left-hand side graph is a tree.) Classical term rewriting theory allows rules such as

$$\text{If}[x\,y\,y]\Rightarrow y.$$

The meaning of the left-hand side is that it will match any application of If whose second and third arguments are textually identical. For other applications of term rewrite systems, non-left-linear rules have their uses, but textual equality of perhaps unevaluated subterms is an unsatisfactory concept to include in a programming language. In addition, the semantics of term rewriting and graph rewriting deviate significantly in the presence of non-left-linearity [3]. None of the languages we have mentioned includes it. In Miranda one may write apparently non-left-linear rules (e.g. if $x\,y\,y = y$), but this is merely syntactic sugar for a use of the equality operator, something quite different from textual equality.

4.5. The rewrite system must be an operator-constructor system

We require that an operator (i.e. the principal function symbol of some rule) may not appear on the left-hand side of any rule except as its principal function symbol. Miranda, Hope and ML all make a similar restriction. It means, essentially, that the pattern-matching performed by a rule for an operator cannot "look inside" an application of another operator.

4.6. The rewrite system must be non-ambiguous

It must be impossible for a term to match two rules for the same operator. Laville's transformation [16] removes ambiguities which would otherwise be resolved by rule-ordering. We require that there be no other ambiguities. An example of what is disallowed is:

$$\text{Or}[\text{True}\ x]\Rightarrow\text{True}|$$

$$\text{Or}[x\ \text{True}]\Rightarrow\text{True}.$$

The term Or[True True] would match both of these rules. This condition, together with the preceding two, implies that the system is regular.

Regularity is an important concept in the theory of term rewriting. When a redex is reduced in a term t of a regular TRS, every subterm of t gives rise, in a simple and intuitive sense, to a (possibly empty) set of similar subterms in the resulting term; its *descendants*. The descendants of a redex are redexes by the same rewrite rule. This leads to a simple proof of the Church-Rosser property of regular TRSs,

and to the concept of *neededness*. In a regular TRS, a subterm t of a term t' is *needed* (*head-needed*) by t' if t' cannot be reduced to normal (constructor) form without reducing t to constructor form.

4.7. The rewrite system must be strongly sequential

This means that for every term T of the system, not in normal form, it must be possible to find at least one redex T' contained in T, such that every reduction of T to normal form must reduce T'; and furthermore, such a redex can be determined without knowledge of the right-hand sides of the rules.

A formal definition of strong sequentiality for regular TRSs can be found in [14, 18]. The definition is rather complicated. However, our restriction to operator-constructor systems leads to a great simplification.

Definition 4.1. Given a TRS T containing an operator F, a *partial F-redex* is a linear term t such that the left-hand side of at least one rule for F is an instance of t. A partial F-redex is *proper* if it is not an F-redex.

A regular operator-constructor TRS is *strongly sequential* if for every operator F of the TRS, and every proper partial F-redex t, there exists a variable x of t such that every left-hand side t' of a rule for F which is an instance of t instantiates x to a non-variable. Such a variable is an *index* of t.

Note that this condition can be checked for each operator independently, and depends only on the left-hand sides of the rules. If we say that an operator is strongly sequential if its set of rules is, then a TRS is strongly sequential iff each of its operators is.

Example 4.2. The classic example of a non-sequential rewrite system is "Berry's F" [5]:

$$F[x\,0\,1]\Rightarrow\ldots|$$

$$F[1\,x\,0]\Rightarrow\ldots|$$

$$F[0\,1\,x]\Rightarrow\ldots\ .$$

These rules are regular. Consider the proper partial F-redex $F[x\,y\,z]$. For each variable of this term, there is a rule for F whose left-hand side is an instance of this term but which does not instantiate the given variable.

The non-sequentiality of these rules is reflected in the difficulty of using them to evaluate terms lazily. Suppose that we are given a term $F[t_1\,t_2\,t_3]$. If t_2 and t_3 evaluate to, respectively, 0 and 1 (where we assume that in addition to the rules for F there are some typical rules for e.g. arithmetic), then we know that only the first rule for F can apply, no matter what t_1 evaluates to, even if t_1 has no normal form at all. But similarly, even if t_2 has no normal form, we might (so long as we do not try to evaluate t_2) discover that the term matches rule 2, and if t_3 has no normal

form, it might be that the term matches rule 3. Whichever of t_1, t_2, or t_3 we try to evaluate first, we may fall into a non-terminating computation which we might have avoided if we had chosen a different subterm to evaluate. We do not know which of t_1, t_2, or t_3 we must evaluate, without knowing in advance which of them have normal forms. But knowing whether a term has a normal form is a fundamentally undecidable problem [2].

Contrast the following system:

Shorter[x Nil]⇒False|

Shorter[Nil Cons[y ys]]⇒True|

Shorter[Cons[x xs] Cons[y ys]]⇒Shorter[xs ys].

These rules compare the lengths of two lists. Given a term Shorter[t_1 t_2], we clearly must evaluate t_2 until it turns into the term Nil or a term of the form Cons[t t']. In the latter case (and only then) we must evaluate the term t_1 to the same extent, to find out whether the second or the third rule applies.

The crucial properties of strong sequentiality are that it is a decidable property of regular TRSs, and that for any strongly sequential TRS one may construct an evaluation algorithm which will find the normal form of every term which has one. Such an algorithm is given in [14, 18].

4.8. Correctness

Let $t = F[\underline{t}]$ be an operator term. Then we require that if every operator subterm of t, other than t itself, is replaced by a new variable, the resulting term (the *constructor prefix* of t) must be unifiable with the left-hand side of some rule for F.

In English, this means that pattern-matching may only fail because of unevaluated subterms. Most functional languages consider it an error to make an application of an operator to constructor terms not matching any of the operator's rules. Thus we are assuming that the user has written a correct program.

Definition 4.3. A term rewrite system which satisfies the restrictions of this section is a *simple* term rewrite system. When read as a TGRS, it is a *simple* TGRS.

5. Translating simple term rewrite programs into Dactl

A term rewrite system written in Dactl syntax is already a legal Dactl program. Dactl rewriting by definition acts identically to term graph rewriting, except for the control markings. However, without control markings, no rewriting will be performed. The task of translating a term rewrite system into Dactl thus amounts to deciding on control markings for the right-hand sides which will lead Dactl execution to the normal form. The choice of control markings will be guided by requiring certain invariants to hold throughout the execution of the resulting program.

Invariant 1. When a node bearing an operator is active, its arguments are already sufficiently evaluated for one of the rules for that operator to match.

Invariant 2. The number of # marks on a node equals the number of its notify out-arcs.

Invariant 3. Every notify arc points to an active or blocked node.

Invariant 4. Every active or blocked node is head-needed.

Invariant 1 ensures that when a node is deactivated, it will be in constructor form. Invariants 2 and 3 prevent certain forms of starvation. Invariant 4 prevents unnecessary computation.

This only ensures that the initial graph is evaluated to constructor form, not to normal form. We shall attend to that later; in the meantime our primary concern will be to assign control markings to the rules which will force active nodes to be evaluated to constructor form.

We first describe a simple-minded algorithm. It will maintain invariants 2, 3 and 4, but in general violates 1. For a certain subclass of the simple systems, the *flat* systems, it will maintain invariant 1; non-flat systems will be handled by transforming them into flat ones.

Marking algorithm

Mark the root of the right-hand side of every rule with "*" or "#", the choice depending on its function symbol. Mark an identifier, constructor, or non-pattern-matching operator with "*". Mark a pattern-matching operator with "#", where n is the number of its always-matched places. Mark each arc to an always-matched argument with a notify mark, and apply the marking algorithm recursively to the nodes pointed to by those arcs.

End of marking algorithm

6. Reaching normal form

The translation we have described will only reduce active nodes to constructor form. To obtain normal forms we must add rules defining a normal form evaluator. For a program whose only constructor symbols are integers, Nil and binary Cons, the following rules suffice:

$$\text{Eval}[x{:}\text{INT}] \Rightarrow {}^*x \,|$$

$$\text{Eval}[x{:}\text{Nil}] \Rightarrow {}^*x \,|$$

$$\text{Eval}[\text{Cons}[x\,y]] \Rightarrow \#\#\text{Cons}[\hat{}\#\text{Eval}[\hat{}{}^*x]\ \hat{}\#\text{Eval}[\hat{}{}^*y]].$$

For every other constructor symbol used in the program, a similar rule for Eval would be required. The initial rule of the program would then be written as:

$$\text{Initial} \Rightarrow \#\text{Eval}[\hat{}(\text{term activated by the marking algorithm})]$$

7. Simple example

Term rewrite rules:

AddMatrix[Nil]⇒0|

AddMatrix[Cons[*x y*]]⇒IAdd[AddList[*x*] AddMatrix[*y*]]|

AddList[Nil]⇒0|

AddList[Cons[*x y*]]⇒IAdd[*x* AddList[*y*]].

We assume that there are also rules for the integer addition operator IAdd. Dactl rules:

AddMatrix[Nil]⇒*0|

AddMatrix[Cons[*x y*]]⇒##IAdd[^#AddList[^**x*] ^#AddMatrix[^**y*]]|

AddList[Nil]⇒*0|

AddList[Cons[*x y*]]⇒##IAdd[^**x* ^#AddList[^**y*]].

8. Flattening transformations

We can distinguish three different ways in which the marking algorithm may violate invariant 1:

(i) The rules for an operator may require to know more than merely the principal function symbols of its pattern-matched arguments. Guaranteeing that its needed arguments are reduced to constructor form before the operator node becomes active may not ensure that a rule then matches.

(ii) Different rules for an operator may pattern-match different places. It cannot be known in advance which rule will eventually match some use of an operator, and so it may not be known which arguments should be activated.

(iii) An operator node $F[t_1 \ldots t_n]$ may appear on a right-hand side, in a position which the marking algorithm does not reach. It will be left unmarked. If it later becomes active, its arguments may be insufficiently evaluated for pattern-matching to succeed.

We handle these possibilities by transforming offending systems to eliminate them.

8.1. Multi-level patterns

Consider the following rules.

Second[Nil]⇒Error|

Second[Cons[*x* Nil]]⇒Error|

Second[Cons[*x* Cons[*y z*]]]⇒*y*|

F[*x y*]⇒Second[*x*].

The marking algorithm will mark the last rule thus:

$$F[x\,y] \Rightarrow \#\text{Second}[\hat{}{*}x].$$

However, when a reduct of this rule is evaluated, the subgraph corresponding to x will only have been evaluated far enough to obtain a node of the form Nil or Cons$[t_1\ t_2]$. In the latter case, if t_2 is not in constructor form, then none of the rules for Second will match. Evaluating x to constructor form is not enough, as the rules for Second require more knowledge of its argument than the principal function symbol.

We can transform the rewrite system to eliminate such "multi-level" pattern-matching. Define the *level* of a pattern by:

$$level(x) = 0.$$

$$level(F) = 1.$$

$$level(F[t_1 \ldots t_n]) = 1 + \text{the maximum of } level(t_1), \ldots, level(t_n).$$

The second and third rules of the example have level 3. The marking algorithm fails to preserve invariant 1 for rules whose level is greater than 2. We must somehow eliminate "deep" patterns.

In the example, let R_1, R_2 and R_3 be the three rules for Second. Their left-hand sides have levels 2, 3 and 3, respectively. Leave R_1 unchanged. "Trim" the left-hand sides of R_2 and R_3 to level 2, by replacing deeper subterms by new variables, to obtain respectively Second[Cons[$x\,x1$]] and Second[Cons[$x\,x2$]]. These being equivalent up to renaming of variables, we replace R_2 and R_3 by a single rule for Second:

$$\text{Second}[\text{Cons}[x\,x1]] \Rightarrow \text{Second1}[x\,x1]$$

where Second1 is a new function symbol. This rule does the pattern-matching common to both R_2 and R_3, down to level 2. Add two rules for Second1, which do the rest of the pattern-matching:

$$\text{Second1}[x\ \text{Nil}] \Rightarrow \text{Error}|$$

$$\text{Second1}[x\ \text{Cons}[y\,z]] \Rightarrow y.$$

These rules have level 2, so we stop here. If R_2 and R_3 had been deeper, the rules for Second1 would have level greater than 2, but one less than the levels of R_2 and R_3, and we would apply the algorithm repeatedly.

The Dactl translation is then:

$$\text{Second}[\text{Nil}] \Rightarrow {*}\text{Error}|$$

$$\text{Second}[\text{Cons}[x\,x1]] \Rightarrow \#\text{Second1}[x\ \hat{}{*}x1]|$$

$$\text{Second1}[x\ \text{Nil}] \Rightarrow {*}\text{Error}|$$

$$\text{Second1}[x\ \text{Cons}[y\,z]] \Rightarrow {*}y|$$

$$F[x\,y] \Rightarrow \#\text{Second}[\hat{}{*}x]$$

Transformation 1: *elimination of rules of level* >2

For any term t, define t/i $(i \geq 0)$ by:

$$x/i = x,$$

$$F[\underline{t}]/0 = y,$$

$$F[\underline{t}]/i+1 = F[\underline{t}/i].$$

Here y is a new variable, not already occurring in the system being transformed. $\underline{t}/i$ is the list of terms obtained by applying $/i$ to every member of $\underline{t}$. t/i always has level $\leq i$, and t is a substitution instance of t/i.

Let F be an operator of the system. Leave unchanged those rules for F whose left-hand sides have level ≤ 2. For each deeper rule R for F, define left'(R) to be the result of replacing, in $left(R)/2$, every argument in a sometimes-matched place of F by a new variable. Partition the deeper rules for F into equivalence classes by the relation:

$$R \sim R' \Leftrightarrow left'(R) \text{ and } left'(R') \text{ differ only in the names of their variables.}$$

Let $R_1, \ldots, R_n$ be one such equivalence class, where R_i is

$$F[\underline{t}_i] \Rightarrow t_i \quad (i{:}1..m).$$

We will replace this equivalence class by a single rule for F and a set of rules for a new function symbol F'. A different F' is chosen for each equivalence class.

Let $F[\underline{t}]$ be left'(R_j) for some $j{:}1..m$ (by the definition of the equivalence it does not matter which). Let $\underline{x}$ be the list of (by linearity, distinct) free variables of $\underline{t}$, in order from left to right. Each $F[\underline{t}_i]$ is equal to $F[\sigma_i(\underline{t})]$ for some substitution σ_i defined on $\underline{x}$. The equivalence class is replaced by:

$$F[\underline{t}] \Rightarrow F'[\underline{x}]$$

$$F'[\sigma_i(\underline{x})] \Rightarrow t_i \quad (1 \leq i \leq n).$$

Strong sequentiality implies that $\underline{t}$ cannot be just a list of variables, and hence that $F'[\sigma_i(\underline{x})]$ has lesser level than $F[\underline{t}_i] = left(R_i)$. Repeating the transformation often enough will therefore eliminate all deep patterns.

End of transformation 1

For the remaining transformations, we assume that this transformation has first been carried out as often as possible.

8.2. Sometimes-matched arguments

In the rules for Shorter stated in Section 4.7, Shorter certainly needs its second argument, but whether or not it needs its first depends on the value of its second. In translating to Dactl we thus do not know whether to mark the first argument of Shorter on the right-hand side of the last rule. The marking algorithm leaves it unmarked, giving the markings #Shorter[xs ^$*ys$]. But then if the second argument evaluates to Cons[...], we will need the first argument, which may be unevaluated.

We eliminate sometimes-matched places by applying a "trimming" similar to that of transformation 1. Instead of replacing deep subpatterns by variables, we replace sometimes-matched places in the pattern by variables.

The first rule for Shorter is left unchanged, as it does not pattern-match any sometimes-matched places. The left-hand sides of the second and third rules are trimmed to Shorter[a Cons[y ys]] and Shorter[b Cons[y ys]], respectively. As these differ only in the names of their variables, we replace them by a single rule:

Shorter[a Cons[y ys]]⇒Shorter1[a y ys]

to carry out the always-matched pattern-matching common to the two rules. Shorter1 is a new symbol, whose rules perform the remainder of the pattern-matching:

Shorter1[0 y ys]⇒True|

Shorter1[Cons[x xs]y ys]⇒Shorter[xs ys].

There are no longer any sometimes-matched places. The marking algorithm gives

Shorter[x Nil]⇒*False|

Shorter[a Cons[y ys]]⇒#Shorter1[^*a y ys]|

Shorter1[0 y ys]⇒*True|

Shorter1[Cons[x xs]y ys]⇒#Shorter[xs ^*ys].

Transformation 2: elimination of sometimes-matched arguments

Let F be a function symbol with arity n. Suppose arguments $1 \ldots a$ of F are always-matched, arguments $a+1 \ldots b$ never-matched, and arguments $b+1 \ldots n$ sometimes-matched, with $n \geq b+1$. (For simplicity, we suppose this convenient ordering of the places of F. It is obvious how the algorithm should be modified for the general case.)

Leave unchanged those rules for F (if any) which pattern-match none of the places $b+1 \ldots n$. Partition the remaining rules into equivalence classes by the relation:

$R \sim R' \Leftrightarrow$ for i: $1 \ldots a$,

$left(R) \cdot i$ and $left(R') \cdot i$ differ only in the names of their variables.

Let one such class be $R_1, \ldots, R_m$, where R_j is

$F[\underline{a}_j \underline{n}_j \underline{s}_j] \Rightarrow t_j$

where $\underline{a}_j$, $\underline{n}_j$ and $\underline{s}_j$ have lengths a, b and c, respectively. $\underline{a}_i$ is a list of non-variables, $\underline{n}_i$ is a list of the same $\underline{n}$. Invent a new function symbol F' (a different function symbol for each equivalence class), and replace $R_1, \ldots, R_m$ by one rule for F and m rules for F':

$F[\underline{a}\ \underline{n}\ \underline{x}] \Rightarrow F'[\underline{y}\ \underline{n}\ \underline{x}]$

$F'[\underline{y}\ \underline{n}\ \underline{s}_j] \Rightarrow t_j \quad (1 \leq j \leq m)$.

Here $\underline{x}$ is a list of new variables the same length as each $\underline{s}_j$, and $\underline{y}$ is the list of all (by linearity, distinct) variables occurring in $\underline{a}$, in order from left to right.

After doing this for all the equivalence classes, F has no sometimes-matched places. Strong sequentiality implies that each F' will have fewer sometimes-matched places than F had. Therefore we can repeat the transformation to eliminate all sometimes-matched places from the TRS.

End of transformation 2

8.3. Masked subterms

Assume that transformations 1 and 2 have been carried out as much as possible. Suppose the right-hand side of some rule contains an occurrence of an operator as an argument to a constructor, or as a never-matched argument to an operator. For example:

$$F[A] \Rightarrow \ldots |$$

$$G[x] \Rightarrow \text{Cons}[F[x]\ \text{Nil}].$$

The marking algorithm will give these markings to the second rule:

$$G[x] \Rightarrow *\text{Cons}[F[x]\ \text{Nil}].$$

The application of F is inactive; if in the course of the computation this node becomes active (as a result of executing some other rule, such as $\text{Head}[\text{Cons}[x\,y]] \Rightarrow *x$), then those arguments which it pattern-matches may not yet be in constructor form, and the rule for F will not match.

Note that there is no problem if the rules for F do no pattern-matching, for then any F node would be a redex without requiring any evaluation of its arguments. There is only a problem when a pattern-matching operator occurs at the root of a subterm which the marking algorithm does not mark. We call such a subterm a *masked* subterm.

We eliminate masked subterms from the system, by ensuring that we never build an application of a pattern-matching operator until it is known to be needed.

In the example, replace the masked subterm $F[x]$ by $F'[x]$, where F' is a new function symbol:

$$G[x] \Rightarrow \text{Cons}[F'[x]\ \text{Nil}].$$

Then add a rule for F':

$$F'[x] \Rightarrow F[x].$$

F' does no pattern-matching, so its use is not masked. The marking algorithm then gives:

$$G[x] \Rightarrow *\text{Cons}[F'[x]\ \text{Nil}]|$$

$$F'[x] \Rightarrow \# F[\hat{}*x].$$

The purpose of F' is to delay constructing an application of F until it is known to be needed.

Transformation 3: elimination of masked subterms

Let t be the right-hand side of a rule R, and let $t' = F[\underline{t}]$ be a maximal masked subterm of t. That is:

(i) F is a pattern-matching operator;

(ii) applying the marking algorithm to t does not mark the root of t';

(iii) it does mark the parent of t'.

We invent a new operator F', replace every use of F as the principal symbol of a maximal masked subterm by F', and add the following rule for F':

$$F'[\underline{x}] \Rightarrow F[\underline{x}]$$

where $\underline{x}$ is a list of distinct variables the same length as $\underline{t}$. (Alternatively, if some existing operator F' has a rule of this form, then we use that operator instead of adding a new one.)

The new rule does no pattern-matching, so no use of F' is masked. The right-hand side of the new rule contains no masked subterms. Thus we have reduced the number of masked subterms in the system by one. Repeating the transformation will eliminate all masked subterms.

End of transformation 3

For each of three transformations, the set of terms of the resulting system is defined to be the closure of the set of terms of the original system by the reduction relation of the new system and the subterm relation.

9. Correctness of the Dactl translation for flat TRSs

Definition 9.1. A simple TRS is *flat* if:

(1) every left-hand side has level no more than 2;

(2) no operator has any sometimes-matched places;

(3) there are no masked subterms.

Theorem 9.2. *Let $\mathbf{R}$ be a flat TRS, and t a closed term of $\mathbf{R}$. Let $\mathbf{D}$ be the Dactl program resulting from applying the marking algorithm to $\mathbf{R}$, and adding the rule:*

$$\text{Initial} \Rightarrow \{t \text{ marked by the marking algorithm}\}$$

(Braces enclose Dactl comments; we are using them here as meta-comments.) If $\mathbf{D}$ terminates, it terminates with a graph in constructor form containing no markings, to which t can be reduced in $\mathbf{R}$. If $\mathbf{D}$ does not terminate, then t has no constructor form in $\mathbf{R}$.

Proof (Outline). By the results of [3], we may assume that t has a normal form in $\boldsymbol{R}$ iff t has a normal form in the corresponding TGRS $\boldsymbol{R}^{\mathrm{G}}$. Since Dactl computation is closer to the latter system, it is simplest to study the relation between $\boldsymbol{D}$ and $\boldsymbol{R}^{\mathrm{G}}$. It is routine to check that each step of $\boldsymbol{D}$, whether it is a rewrite or a deactivation, preserves the invariants listed in Section 5, and the starting graph *Initial clearly satisfies them. Those invariants, and the simplicity of $\boldsymbol{R}^{\mathrm{G}}$, imply that $\boldsymbol{D}$ reduces only head-needed redexes, and always finds one to reduce if one exists. From [3], head-needed reduction in a TGRS finds a head normal form of any term graph which has one. □

Theorem 9.3. *Let $\boldsymbol{R}$ be a flat TRS, and t a closed term of $\boldsymbol{R}$. Let $\boldsymbol{D}$ be the Dactl program resulting from applying the marking algorithm to $\boldsymbol{R}$, and adding the rules for* Eval *and* Initial *described in Section* 6. *If $\boldsymbol{D}$ terminates, then t has a normal form in $\boldsymbol{R}$, and $\boldsymbol{D}$ terminates with that normal form, with no control markings. If $\boldsymbol{D}$ does not terminate, then t has no normal form in $\boldsymbol{R}$.*

Proof. The previous theorem shows that if t has a constructor form $F[\underline{t}]$ then $\boldsymbol{D}$ begins with

$$\text{Initial} \Rightarrow \#\text{Eval}[\hat{}\{\text{marked version of } t\}]$$
$$\Rightarrow^{*} *\text{Eval}[F[\underline{t}]].$$

Then an Eval rule applies, to invoke evaluation of the components of $\underline{t}$. By induction, the whole normal form of t is eventually computed, if it exists; otherwise, there will be a non-terminating computation. □

10. Correctness of the flattening transformations

We need to know that a TRS resulting from flattening a simple TRS in some sense implements it. For general TRSs, it is difficult to find a simple definition of this notion, and several have appeared in the literature. However, if one bears in mind the intended use of the simple TRSs considered here, the following definition seems appropriate.

Definition 10.1. Let T and T' be simple TRSs. T' *implements* T iff:

(i) T' contains all the function symbols and terms of T;

(ii) every normal form of T is a normal form of T';

(iii) the relations $\rightarrow_T^*$ and $\rightarrow_{T'}^*$ coincide on terms of T.

Thus if T' implements T, and $t \in \text{Terms}(T)$ has a T-normal form, then that normal form can be found by T'-reduction; and any T' reduction starting from t can be completed to a T'-reduction to the T-normal form of t (because T', being simple, is confluent). It is clear that "implements" is a reflexive and transitive relation.

Given TRSs T and T' where T is simple, define $T \rightarrow_i T'$ ($i = 1$, 2, or 3) to mean that T' is obtained from T by one application of the transformation i.

Theorem 10.2. *If T is simple, then performing transformation* 1 *as often as possible, then transformation* 2, *then transformation* 3, *will transform T into a flat TRS which implements T.*

Proof (Outline). From the description of the transformations, it is clear that the sequence of transformations takes T through a sequence of simple TRSs, and terminates with a flat TRS. We need only check that each transformation in the sequence transforms the current TRS into a TRS which implements it.

Only condition (iii) of the implementation property requires much work, and is proved by arguments based on the possibilities for reordering the elements of a reduction sequence in a regular TRS. The proof is greatly simplified by moving to term graph rewrite systems and using the relations proved in [3] between TRSs and TGRSs. □

As a corollary, we find that it is not necessary to check strong sequentiality before attempting to flatten a TRS. If it satisfies all the other conditions of simplicity but is not strongly sequential, then some application of transformation 1 or transformation 2 will fail to make progress, by producing a pattern-matching operator with no always-matched places.

The counting arguments used to prove termination also lead to an estimate of the complexity of the flattening transformation. To define such an estimate we must first give a measure of the size of a TRS. We take the size of TRS to be the sum of the sizes of its rules, the size of a rule to be the sum of the sizes of the left- and right-hand sides, plus 1, and the size of a term to be the number of nodes in its syntax tree. The size of something is denoted by $\| \ldots \|$.

Theorem 10.3. *A simple TRS T can be transformed to a flat TRS T' such that* $\|T'\| = O(\|T\| + \sum \{\|\mathit{left}(R)\|^2 \mid R \in \mathrm{Rules}(T)\})$.

The quadratic component results from transformation 1, and is only significant when dealing with left-hand sides which are both very large and have very unbalanced parse-trees. In practice it is unlikely to be important. Note that the sizes of the right-hand sides have little influence on the growth due to flattening. Transformations 1 and 2 do not affect the original right-hand sides. Transformation 3 may break them up into smaller pieces, but does not make copies of them. The extra size it introduces is proportional to the total of the arities of all the masked subterms it finds.

11. Implementation

We have written a translator from Clean to Dactl using all these transformations. The resulting Dactl programs have been run by a Dactl compiler on a Sun.

There is not yet a consensus on a good set of benchmark problems to evaluate the speed of a functional language implementation. The well-known NFib function, defined by

NFib[0] = 1 |

NFib[1] = 1 |

NFib[*n*:(INT − 0 − 1)] = IAdd[1 IAdd[NFib[ISub[*n* 1]] NFib[ISub[*n* 2]]]]

(the functional equivalent of, say, counting up to 2^{31}) is so simple that applying the translator to a definition in Clean yields Dactl essentially identical to a hand-coded version. Some larger benchmarks have been studied: list-reversing, and the small microprocessor simulator described in [9, 10]. In all the cases we have studied, the automatic translation runs no more than a few percent slower than a hand-coded Dactl program to do the same job.

12. Some optimisations and extensions

The algorithms presented here illustrate the general approach, but many optimisations can be made. Some, such as strictness analysis and symbolic evaluation, are techniques commonly built-in to functional language implementations. Here we will see that they can be expressed in Dactl.

12.1. Do not activate constructors

The marking algorithm activates more things than is necessary. No constructor need be made active on the right-hand side of a rule except at the root. The version of factorial stated in Section 2 makes this optimisation.

12.2. Taking graphs seriously

If, as is argued for in [3], one views term graph rewriting as a computational model in its own right, rather than as an implementation of term rewriting, then some extra features may profitably be included in the languages to be translated. For example, both ML and Clean include a syntax allowing interior nodes of the left-hand side of a rule to be named, and those names to be used on the right-hand side. Thus, instead of writing a rule like

Last[Cons[*x* Cons[*y z*]]]⇒Last[Cons[*y z*]]

we may instead write

Last[Cons[*x w*:Cons[*y z*]]]⇒Last[*w*].

The right-hand side of a rule may be a cyclic graph, as with this "knot-tying" fixed-point operator

Fix[*f*]⇒*y*:Apply[*f y*].

The flattening and marking algorithms are easily extended to handle such features.

12.3. Strictness analysis for increased parallelism

Strictness analysis, such as described in [19], can discover within the right-hand sides, needed subterms which may not be reached by the marking algorithm (which is itself a rudimentary form of strictness analysis). For example:

Last[Nil]⇒Error|

Last[Cons[x Nil]]⇒x|

Last[Cons[x w:Cons[y z]]]⇒Last[w]

Translating this TRS to Dacti gives:

Last[Nil]⇒*Error|

Last[Cons[x y]]⇒#Last1[x ^*y]|

Last1[x Nil]⇒*x|

Last1[x w:Cons[y z]]⇒*Last[w].

We can see that when the second rule for Last1 matches, the term matched by z will eventually be evaluated. We might as well activate it immediately, resulting in a translation for the last rule which obtains more parallelism:

Last1[x w:Cons[y z]]⇒*Last[w], *z.

12.4. Omit unnecessary variables

In the example of Section 8.2, the second argument to Shorter1 is never used. The translation can be simplified to:

Shorter[x, Nil]⇒*False|

Shorter[a Cons[y ys]]⇒#Shorter1[^*a ys]|

Shorter1[Nil ys]⇒*True|

Shorter1[Cons[x xs] ys]⇒#Shorter[xs ^*ys].

12.5. Reduce needed redexes present in right-hand sides

The right-hand side of the rule for Last1 at the end of Section 12.3 contains an active term which is already a redex. We might as well reduce it in the rule itself, instead of reducing a copy every time the rule is applied. Replace the rule by

Last1[x w:Cons[y z]]⇒#Last1[y ^*z].

This is a form of symbolic evaluation. Although the given system may contain few such redexes, the flattening transformations often introduce more, and it is worth reducing them in place.

13. Further developments

Dactl's fine-grain control allows the definition of normalising evaluators other than that described in Section 6. For example, "early completion" semantics can be achieved by rules such as

$$\text{Early}[\text{Cons}[x\ y]] \Rightarrow *\text{Cons}[\#\text{Early}[\hat{}*x]\ \#\text{Early}[\hat{}*y]].$$

A computation evaluated in this way will return a result as soon as constructor form is reached, while continuing the computation of the rest of the normal form in parallel. Alternatively, instead of interposing the evaluator between the node to be evaluated and its parents, we can have a right-hand side of the form:

$$\ldots z \ldots, \#\text{ParEval}[\hat{}*z]$$

and rules such as

$$\text{ParEval}[\text{Cons}[x\ y]] \rightarrow \#\text{ParEval}[\hat{}*x], \#\text{ParEval}[\hat{}*y].$$

The single arrow indicates that no redirection is performed; the purpose of executing the rule is simply to cause evaluation of x and y. If we only know that at least the "backbone" of a list will eventually be required, we can define a backbone evaluator:

$$\text{Backbone}[\text{Cons}[x\ y]] \rightarrow \#\text{Backbone}[\hat{}*y].$$

Many other variations are possible. Such evaluators can be used to implement the more sophisticated methods of strictness analysis described in [7].

Dactl's computational model allows multiple rewrites to be performed concurrently on a graph which is shared among all the processing agents. The semantics stipulates that concurrent rewrites must not interfere with each other: a Dactl computation, however it is actually performed, must have the same effect as some sequence of rewrites. This basic assumption is necessary in order to be able to reason about Dactl computations as we have done in this paper. It might appear at first glance that a real parallel implementation would require heavy locking protocols to prevent interference between agents. However, we have shown in [15] that the Dactl programs produced by the translation described here can be implemented without any such overhead, on hardware whose atomic, serialisable, actions are only capable of accessing single nodes.

14. Conclusions

We have shown how a class of term rewrite systems may be translated into a low-level rewrite language in which the evaluation strategy is represented explicitly. By ensuring pre-evaluation of the needed arguments to every application of an operator, the behaviour of the resulting system has much in common with data-flow

implementations. However, in contrast to data-flow, the translation retains lazy semantics. Little efficiency is lost, relative to hand-coded Dactl.

Dactl is at a sufficiently low level to allow the expression, not just of the reduction relation of a rewrite system, but also of common techniques for implementing the reduction.

Acknowledgment

This work was partially supported by the Alvey programme. It grew out of work with colleagues in the Declarative Systems Project at the University of East Anglia, and in the Dutch Parallel Reduction Machine Project at the University of Nijmegen.

References

[1] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21** (1978) 613–641.

[2] H.P. Barendregt, *The Lambda Calculus* (North-Holland, Amsterdam, 1984).

[3] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep, Term Graph Rewriting, in: *Proc. PARLE Conference*, Lecture Notes in Computer Science **259** (Springer, Berlin, 1987) 141–158.

[4] J.A. Bergstra and J.W. Klop, Conditional rewrite rules: confluency and termination, Report IW 198/82, Stichting Mathematisch Centrum, Amsterdam, 1982.

[5] G. Berry, Stable models of typed lambda-calculi, in: G. Ausiello and C. Böhm, eds., *Proc. 5th Internat. Conf. on Automata, Languages, and Programming*, Lecture Notes in Computer Science **62** (Springer, Berlin, 1978).

[6] T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer and M.J. Plasmeijer, Clean: a language for functional graph rewriting, Internal report 95, Computing Science Department, University of Nijmegen, 1987.

[7] G. Burn, Evaluation transformers—a model for the parallel evaluation of functional languages, in: *Proc. 3rd Internat. Conf. on Functional Programming Languages and Computer Architectures* (1987).

[8] R.M. Burstall, D.B. MacQueen and D.T. Sannella, HOPE: an experimental applicative language, Report CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[9] A.R. Clare, Evaluating declarative programming, in: *Proc. Alvey Technical Conference* (IEE, London, 1988).

[10] A.R. Clare, Evaluating declarative programming, Ph.D. thesis, School of Information Systems, University of East Anglia, 1988.

[11] J.R.W. Glauert, J.R. Kennaway and M.R. Sleep, Dactl: a computational model and compiler target language based on graph reduction, *ICL Tech. J.* **5** (1987) 509–537.

[12] J.R.W. Glauert, J.R. Kennaway, M.R. Sleep and G.W. Somner, Final specification of Dactl, Report SYS-C88-11, School of Information Systems, University of East Anglia, 1989.

[13] J.R.W. Glauert, Introduction to Dactl, in: *Proc. Alvey Technical Conference* (IEE, London, 1988).

[14] G. Huet and J.J. Lévy, Call by need computations in non-ambiguous term rewriting systems, Report 359, IRIA, 1979.

[15] J.R. Kennaway, The correctness of an implementation of functional Dactl by parallel rewriting, in: *Proc. Alvey Technical Conference* (IEE, London, 1988).

[16] A. Laville, Implementation of lazy pattern matching algorithms, in: H. Ganzinger, ed., *Proc. 2nd European Symp. on Programming*, Lecture Notes in Computer Science **300** (Springer, Berlin, 1988).

[17] R. Milner, A proposal for standard ML, in: *Proc. ACM Conf. on Lisp and Functional Programming* (1984) 184–197.

[18] M.J. O'Donnell, *Equational Logic as a Programming Language* (MIT Press, Cambridge, MA, 1985).

[19] S.L. Peyton-Jones, *The Implementation of Functional Programming Languages* (Prentice Hall, Englewood Cliffs, NJ, 1987).

[20] D.A. Turner, Miranda: a non-strict language with polymorphic types, in: J.P. Jouannaud, ed., *Proc. Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201** (Springer, Berlin, 1985).